10/022787

save for Amendment A send

US-PAT-NO: 6115715

DOCUMENT-IDENTIFIER: US 6115715 A
See image for Certificate of Correction

TITLE: Transaction management in a configuration database

---------- KWIC ---------

Brief Summary Text - BSTX (12):

In a preferred embodiment, the locking mechanism determines whether the transaction attempting to obtain the lock is a blocking or non-blocking transaction, which will determine whether the transaction's thread will be placed in a queue to wait for the lock to be released. In another preferred embodiment, a new node is added to the configuration database if data unrelated to an existing node is created by the transaction, and an existing node is updated if data created by the transaction is related to the existing node. In yet another preferred embodiment, if the transaction is aborted, the database is reverted to an initial state by counteracting all updates in the transaction that were successfully performed. This is followed by releasing the lock on the node and notifying an event manager that the lock has been released.

transactn threads placed in que.

Detailed Description Text - DETX (24):

If the transaction is nonblocking it will attempt to perform other operations on other parts of the client schema while placing the thread that needs the locked entry in a queue and have it wait until the entry is free. Typically a transaction is non-blocked if there are real-time constraints and it is more efficient to continue doing other operations while a particular thread waits. In the described embodiment, the blocking or non-blocking determination is relevant if an attempt to lock an entry fails. As described above, either the entire transaction or a thread within a transaction waits until a lock is obtained. Thus, if the transaction is blocking, control returns to step 502 where the transaction attempts to acquire a lock on the desired entry, this time after waiting in a queue and being notified by an event manager, described below. If the transaction is non-blocking, the process is done since this indicates that the transaction does not want to wait to acquire a lock.

Detailed Description Text - DETX (25):

Once a lock on a desired entry is obtained thereby either locking a single entry or a sub-tree, a transaction handle is created for the transaction at step 506. This handle object is a unique identifier for the specific transaction that caused the lock. In other preferred embodiments, the transaction handle can be any type of identifier useful in determining which locks were caused by a specific transaction. The application proceeds with

this transaction handle to perform updates on the JSD, specifically on the client JSD. A thread in the transaction is allowed to use the transaction handle, and can call to perform on update on an entry or insert a new entry, such as an entry for a new printer under the Devices namespace. In the described embodiment, each transaction is assigned a single transaction handle. This allows the system to determine, if necessary, that a particular update, modification, or insert, belongs to a particular transaction. A transaction handle object can take the form of a single integer in the described embodiment.

Detailed Description Text - DETX (27):

At step 510 the update or updates making up the single transaction performed at step 508 are committed and a number of events occur. The locks acquired at step 502 are released. Thus, all entries are unlocked. This unlocking operation commits the updates made at step 508. In the described embodiment, the locks are released by examining the transaction handle for each lock. Only those locks that have the correct transaction handle are released, i.e. committed. In other preferred embodiments, other mechanisms for matching locks with a specific transaction can be used, such as maintaining a table of records, where each record represents a specific lock. Once the locks are released, as part of the commit phase of step 510, an asynchronous event notification is performed by posting the lock release to a centralized event manager. In other preferred embodiments, a notice that the transaction has been committed is broadcast to all threads waiting on the nodes that were locked without going through an event manager. In the described embodiment, the event manager is notified by a parent or root node of a sub-tree in the client JSD schema once a lock is released by posting the release to the event manager. In the described embodiment, the current active thread that is releasing the lock does not wait for acknowledgment from the event manager or from any of the threads waiting on the node just released. In other preferred embodiments, well-known synchronous methods of event notification can be used to achieve the same result. Any transactions waiting to obtain a lock, regardless of whether exclusive or shared, registers itself with the event manager. The central event manager can then determine which transactions are waiting for the node or nodes that were released. Once the event manager, shown in FIG. 1 as an instance 108 of an Entry 106, is informed that updates or insertions have been committed, it determines which transactions are to be notified. Once the commit has been performed, the transaction is completed at step 512.

Detailed Description Text - DETX (30):

FIG. 6 is a flowchart showing in greater detail step 508 of FIG. 5 in which the update is performed in accordance with one embodiment of the present invention. In order for a transaction to be completed, one or more updates to the configuration database typically must be performed. The update can be changing the contents of an existing node, typically a leaf node, or adding a new node to a sub-tree. Thus, the first step is to perform the actual update to the JSD schema, whether it be the client or server JSD, as shown at step 602. At step 604 an event queue for the transaction is created when the

transaction is instantiated or created. In the described embodiment, each transaction has its own event queue. An event queue can be identified by a transaction handle, which is also created when the transaction is instantiated. In other preferred embodiments, the event queue and transaction handle need not be created precisely when the transaction is instantiated, but at a time shortly thereafter, before updates in that transaction are performed. An event queue is described in greater detail below with respect to FIG. 8. Once an event queue for the transaction is available, all state data relating to a specific update that is necessary to undo the update if necessary is stored as an entry in the event queue at step 606. The state information saved in the event queue at step 606 reflects the state of the configuration database once the update has been performed. Thus, by examining the state data in each entry for a particular transaction, the network will now exactly what state of the database to revert to in the event of a failure. By doing this incrementally for each specific update performed in a transaction, the system can return the configuration database to its original state before the transaction started. This rollback operation is explained further with respect to FIGS. 7 and 8.

Claims Text - CLTX (11):

assigning an identifier to the transaction that caused the lock, wherein the identifier is a unique identifier and is used by a thread in the transaction to perform an operation on a node in the Java system configuration database.

DOCUMENT-IDENTIFIER: US 20020133507 A1

TITLE: Collision avoidance in database replication systems

---------- KWIC ---------

Detail Description Paragraph - DETX (7):

[0025] Collector--an object or process that reads an audit trail, transaction log file, database change queue, or similar structure of a first database, extracts information about specified changes to the first database (e.g., insertions, updates, deletions), and passes that information to the consumer object or process defined below. In Shadowbase.RTM. (a commercially available product made by ITI, Inc., Paoli, Pa.) executing on a COMPAQ NSK (Tandem) source, the collector reads TMF or TM/MP audit trails. In a bidirectional database replication scheme, each of the two databases has an associated collector. The extractor process shown in FIG. 1 of U.S. Pat. No. 5,745,753 (Mosher, Jr.) assigned to Tandem Computers, Inc is similar in operation to the collector.

Detail Description Paragraph - DETX (9):

[0027] Consumer--an object or process that takes messages about database changes that are passed by the collector object or process and applies those changes to the second database. In a bidirectional database replication scheme, each of the two databases has an associated consumer. The receiver process shown in FIG. 1 of Tandem's U.S. Pat. No. 5,745,753 is similar in concept to the consumer, except that the consumer described herein can process multi-threaded (i.e., overlapping) transactions, whereas the receiver process in the Tandem patent cannot process multi-threaded transactions.

Detail Description Paragraph - DETX (10):

[0028] Transaction Receiver--device or object which receives transactions sent by a transaction transmitter for posting to a database. In accordance with the present invention, transaction receivers typically unblock the transaction operations or steps as they are received and apply them into the database. Depending on the nature of the transaction operations or steps, they may be applied in parallel or serially, and the transaction profile may be serialized or multi-threaded (that is, one transaction may be replayed at a time, the transactional order may be altered, and/or the transactions may be replayed in the "simultaneous, intermixed" nature that they occurred in the source database). In one embodiment of the present invention, the transaction receiver is identical to the consumer. In other embodiments, the transaction receiver performs some, but not all, of the functions of the consumer. In a bidirectional database replication scheme, each of the two databases has an associated transaction receiver.

Detail Description Paragraph - DETX (135):

[0153] For reasons of clarity, the descriptions of the present invention describe the application and replication engine as processing one transaction at a time, whereas in a typical implementation these components would be "multi-threaded", that is, able to process many transactions simultaneously.

Detail Description Paragraph - DETX (316):

[0334] The path that tokens take to arrive in the target can be via many routes. The preferred embodiment of the present invention sends them via the audit trail, interspersed at the appropriate point with transaction steps or operations. These tokens can be "piggybacked" onto the last transaction step or operation for their transaction, as well as onto a transaction step or operation for any other transaction. Piggybacking is one preferred scheme in extensive multi-threaded transaction processing environments.